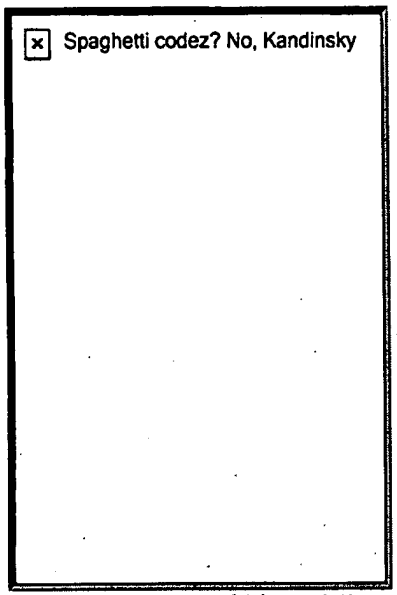
SCRIPTING: Symmetrical sorting (multisort)

Change Background

Spaghetti codez? No, Kandinsky



a **Paul Klee** painting

List of Previously Visited Pages at U.S.

SYMMETRICAL SORTING OF TWO ARRAYS

How to redraw the entries of an array so that it keeps its symmetry with the entries of another array which underwent a sorting process

September 2001

http://web.archive.org/web/20040221101243/http://www.unitedscripters.com/scripts/array3.html#

LIST OF TOPICS DEALT WITH IN THIS DOCUMENT: select one
Main subroutines and how to pass array arguments
Comments to implementation invoking and workings

Symmetrical Sorting Subroutine and Side Subroutines

The main and side routines and how to pass them their arguments

This subroutine (a javaScript so called *multisort*, basically) takes avail of two subroutines - in our case named *highestFirst()* and *highestLast()* - that instruct the sorting process (which applies to number or letter **both**) on whether to sort in decreasing order (higherFirst) or increasing order (higherLast).

In fact the task of the main subroutines is to sort an array either in increasing or decreasing order, and then **reshape another array in such a fashion that it mantains symmetry with the first one. Example:**

```
var testArray1=new Array("one","five","seventy","thirtytwo","two","five")
var testArray2=new Array(1,5,70,32,2,5)
```

As you see in the literal array (*testArray1*) each word is the literal match of the numeral version in testArray2. **What happens if you sort either in ascending or descending order the numeral array?** The symmetry of the first array won't exist any longer.
Whenever you face this problem and you need an array to **keep mirroring the original symmetry it had with another array that underwent a sorting process,** in those cases the soubroutines featured here are exactly what you were in need of. A thorny subject, as you can see.

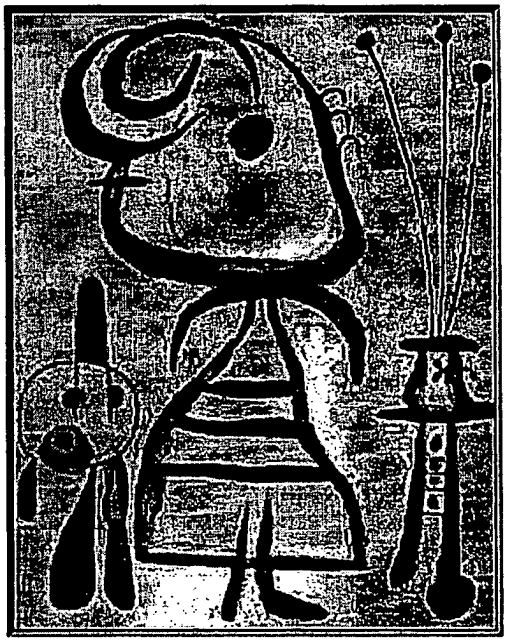
But first the two side subroutines that will help you to make a proper sorting: they must be included in the script:

```
function highestFirst(a,b){
if(a>b){return -1}
else if (a==b){return 0}
else if(a<b){return 1}
}

function highestLast(a,b){
if(a<b){return -1}
else if (a==b){return 0}
else if(a>b){return 1}
}

function highestFirstCI(a,b){
if(a.toLowerCase() > b.toLowerCase()){return -1}
else if (a.toLowerCase() == b.toLowerCase()){return
0}
else if(a.toLowerCase() < b.toLowerCase()){return 1}
}

function highestLastCI(a,b){
if(a.toLowerCase() < b.toLowerCase()){return -1}
else if (a.toLowerCase() == b.toLowerCase()){return
0}
else if(a.toLowerCase() > b.toLowerCase()){return 1}
}
```



The subroutines above will help your sorting process to be performed in the right way, and should be passed as an argument without brackets in within the `sort()` method, such as:
`myArray.sort(highestFirst)` or
`myArray.sort(highestLast)`

1. The first will put the highest value as entry [0], and on from there.
2. The second will put the highest value at the last position.
3. The third would behave as the first, but being *case-**Insensitive***, would also avoid that the capitalized letters would be listed as first (which **would be the default behaviour** of the sorting processes, such as that, for instance, a word starting with a capital Z would appear listed... before a word starting with a lowercase *a*).
4. The fourth is the case insenstive version of #2.

If you do not use any of them, the sorting process would not be complete, since, for instance, number 20 would appear under number two since both start with number 2, but you do not want this: you want number 3 after number 2 and number 20 after number 19!
Just pick the one that best suits your need.

This file deals with symmetrical **sorting**, which is a way of **ordering** a data structure (in this case, an Array): if you need a function that can produce a multisort but not on an ordering process but on a randmo process such as a shuffling of an array, check the **symmetrical shufflings file**.
Here (below) is now the core subroutine. Keep in mind that **the argument "array1" passed to *symmetricalSorting()* must be the array that is meant to be sorted** while **the argument "array2" is the array whose symmetry you want to preserve/rebuild**. Failing remembering this might cause you to sort the wrong array! **The first argument** is the array meant to undergo the sorting, **the second argument** is the array whose symmetry must not be fooled by the sorting of the first one. The third argument affects the type of sorting function used and may be:

- **none or zero**: uses *highestLast()*
- **1** : uses *highestFirst()*
- **2** : uses *highestLastCI()*
- **3** : uses *highestFirstCI()*
- **other** (say **4**) : uses nothing.

THE *symmetricalSorting* CODEX

```
function
symmetricalSorting
(array1, array2, how){
var copy1=new Array();
var copy2=new Array();
var smartDouble=new Array
();
for(var
i=0;i<array1.length;i++){
copy1.length++
copy1[copy1.length-1]
=array1[i]
smartDouble.length++;
smartDouble
[smartDouble.length-1]=0;
};
copy2.length=copy1.length
;
if( !how && window
["highestLast"] )
{copy1.sort(highestLast)}
else if(how==1)
```

Show Codex

The model above is Adriana Skleranikova

HOT **LOADS OF ADRIANA SKLERANIKOVA ON THE NET** HOT

Comments

Hints on how they work and how to implement and invoke them

We first initialize some argument values:

1. Our subroutines will yield ulitmately an array of **two** entries:
   - **FIRST ENTRY [0]** will return the (by now **sorted**) `array1 argument`
   - **SECOND ENTRY [1]** will return the `array2 argument` **redrawn** to exhibit the original symmetry with array1
2. Now we make copies of the arrays: this will grant you that in case you want to leave **unaffected** the originals you can do that.
3. We sort the array1 argument (that is: the first array)
4. We execute calculations on the sorted array (copy1) to see when identical to array1 argument (which is left unaffected since we copied it, so it **still exhibits the original symmetry with array2!**).
5. When identical, we pick the corresponding index entry from array2 and assign it to copy2: this is the trick which keeps the symmetry!
6. In order to avoid to be fooled by possible copies (that is: an array might have two entries with the same name) we devised a way to mark each entry once found its collocation, that is what the `smartDouble` array is for.
7. I suggest to you to use only arrays whose entries you're already sure, thorugh some validation, are never undefined (empty) or carrying values like false or zero which might be interpreted as undefined. Especially Netscape might be induced to read a number like 0 as an undefined entry (workaround smart hint: a parseFloat() on the entry...)
8. Keep in mind that the return is an array of 2 entries so you'd add the index (either `[0]` or `[1]` to the invoking statement; below are two possible syntaxes (both invoking index [1], but you could invoke [0] as well) to invoke these subroutines; the first assuming you want to sort array1 with the lowest value as first, the second assuming you need a sorting process with the highest value first and thus passing also the 3rd argument `how`.

   If you put them in within an `alert()` (for instance providing the two `testArrays` at top of page as we do in the example below) you will see it works perfectly:

```
symmetricalSorting(testArray2,testArray1)[ 1]
//or:
symmetricalSorting(testArray2,testArray1,1)[ 1]
```

SCRIPTING: Binary and Multi Binary Searches

Change Background

Cori Lane

List of Previously Visited Pages at U.S.

BINARY AND MULTI BINARY SEARCH

You may know a binary search is a procedure to scan an Array of even million entries in less than 10 moves, to locate a specific entry. None the less, such search speeds up an Array scanning so dramatically only if the Array has been *previously* sorted, namely has undergone a previous ordering alphabetical process.
My *Multi Binary* can speed up a loop an average 25%, *without* the Array having been sorted at all!

May 2002

http://web.archive.org/web/20040208132443/http://www.unitedscripters.com/scripts/algo6.html#

LIST OF TOPICS DEALT WITH IN THIS DOCUMENT: select one
WHAT A MULTI BINARY IS COMPARED TO A BINARY SEARCH
The CODES and the TEST FORM
Traditional BINARY search JS implementation

The model above is Cori Lane
HOT **LOADS OF CORI LANE ON THE NET** HOT

WHAT A MULTI BINARY IS AS COMPARED TO A BINARY SEARCH

Multi Binary can emulate what a binary search does, but without any need for the Array to have been sorted first!

Let me start stressing one important thing: I see at times programmers besieged by requests by their boss to produce the code asap: as soon as possible.
I'd take my chance here to gently remind you that programming is not a matter of productivity, but mostly a matter of fantasy and patience: wait for an insight, which is sent by fantasy, and which just can't be drawn out by an order: as Jon Bentley points out in his well affirmed *Programming Pearls*:

> «Good programmers are somewhat lazy: they sit back and wait for an insight rather than rushing forward with their first idea»

And for more on this, see also the quote by Bentley at **this section of the current file**.

Now, there are a few ways to loop up the speed of a loop, but all of them should be undertaken only in one case: whenever the Array to be scanned has a length which goes beyond some common limits. In JavaScript Arrays are normally just snippets, but it all widely depends on how much power you put in your scripts; whatever the case, it may not be uncommon to find Arrays of 1,000 entries.
We can say that each process whose aim is to speed up a loop cannot do too many complex things on the Array it scans, for the more the conditional statements and operations you nest in your loop, the slower it can get. Therefore the main thing power loops do is to *locate specific entries*.

Well, my multiBinary is an original approach, meant to speed up loops (as well as another script of mine called *hasher* is, and which can be found clicking here): in fact the classical binary search (and I provide you here with a javaScript implementation of a binary search *as well*, although that is **not** my invention, whereas **multiBinary** is) divides each Array in slices and if it finds at the middle of that slice a range within which the searched for item is located, it shrinks there, otherwise divides by an half again: a process that in less than 10 moves can locate an entry on a one million long object (a similar process is called **insorting**, whereas with an *insorter* instead of locating an entry, you search for the point where you can insert an entry: an *insorter at United Scripters* can be found clicking

here).
All such processes named after either *binary search* or *insorting* heavily rely on the fact the given Array must (absolutely must) have undergone first a sorting process which listed it in alphabetical order: that's the condition upon which a process based on dividing by halves an Array can safely locate an entry.

Conversely, my multiBinary can speed up by an order of 100% or at times even 300% the search for an item on an unsorted Array. It therefore can return either the position index of the first item found matching the given searched value, or an Array which collects all the position indexes where such value has been found in the Array.

My concept was: given an Array whose length is remarkable, say 50,000 entries, you can divide it in a set of smaller segments; the default length of such segments is 50, but you can make them longer or shorter by passing the arguments; in our case, 50 would divide the 50,000 entries long Array into 1000 smaller Objects.
On each of these objects my multiBinary triggers what I call (another relatively cool feature) a double edge zip scan, namely one counter scans its lot of 1000 entries from the top down while another scans it from the bottom up and at most they stop midway (*at most* means: unless they find a match first!); at the same time all the other subsets of 1000 entries would be double edge scanned as well.
No wonder this may make you locate an entry position with a speed dramatically higher than a simple linear loop. Of course, if perchance your searched entry is in the very first dozens leading (head) positions in the 50,000 entries Array, probably a linear loop may appear faster still: but it doesn't appear such any longer if your given entry is at, say, index 37,508.
None the less, the meaningful speed up can be **experienced only if** you search for one single entry; in fact, if you search for **all** the entries that match the given searched value, then the speed up advantage as compared to a *traditional* loop gets lost: in fact in order to collect all the instances of a given value instead than stopping at the first met, the script cannot return the result as soon as it finds one but has to loop the whole array as well as a traditional loop: therefore, its advantage gets lost, for in within each scanned entry, the traditional loop has to do nothing whereas the multiBinary has to update all its counters for all the fragments the input array has been divided into. None the less, it is still faster if it gets run to search only one item (that is exactly, moreover, what mere binary searches search for), but I let the feature which allows you to collect multiple entries as well: better one feature more than one less, after all.

I think it is an interesting approach, which might really live up to its promises only with *quantic* computers though, which are capable of handling simultaneous processes.

You pass to the *multiBinary* script its arguments as follows:

///// ARGUMENTS	
///// array	Just the input array object. if none or it has no length, the function returns null
///// find	The value to search for in the Array, maybe a Number, an Object, a String (in such case in between quotes obviously); if no *find* argument gets passed, the function returns null
///// continuous	If it is not passed or passed as zero, the function returns either: 1. Number if successfully found an item in the Array which matches the *find* argument, and such number obviously represents the index position of the matching item in within the Array 2. null if no match is found. Conversely if *continuous* is passed (for instance as number 1), the function returns either: • Array whose each entry is the numerical index value of the input Array where an instance of the given *find* argument was located. In other words, *continuous* allows you not to let the script stop at the first found instance! • null if no match is found.
	This default value is 50. But if you pass it, the script will divide the input array in segments whose length is equivalent to the given *subset* value. If such value is lower than zero or higher than the amount of available entry, the script would force it to be equal to the

///// subset	**array length (which would boil down the issue to a traditional linear loop).** **Obviously, it is possible the amount of entries cannot be evenly divided by such subsets. In such case the script produces the last subset as a bit lengthier or a bit shorter as much as needed to accommodate the possible remains, and moreover such last snippet would be looped autonomously: I chose so in order to avoid nesting in the main loop (the one parsing *all the other* perfectly evenly divided subsets, which almost necessarily are to be the majority) further additional conditional checks and statements that would have slowed down the whole when parsing long Arrays.**

CODES & TEST FORM

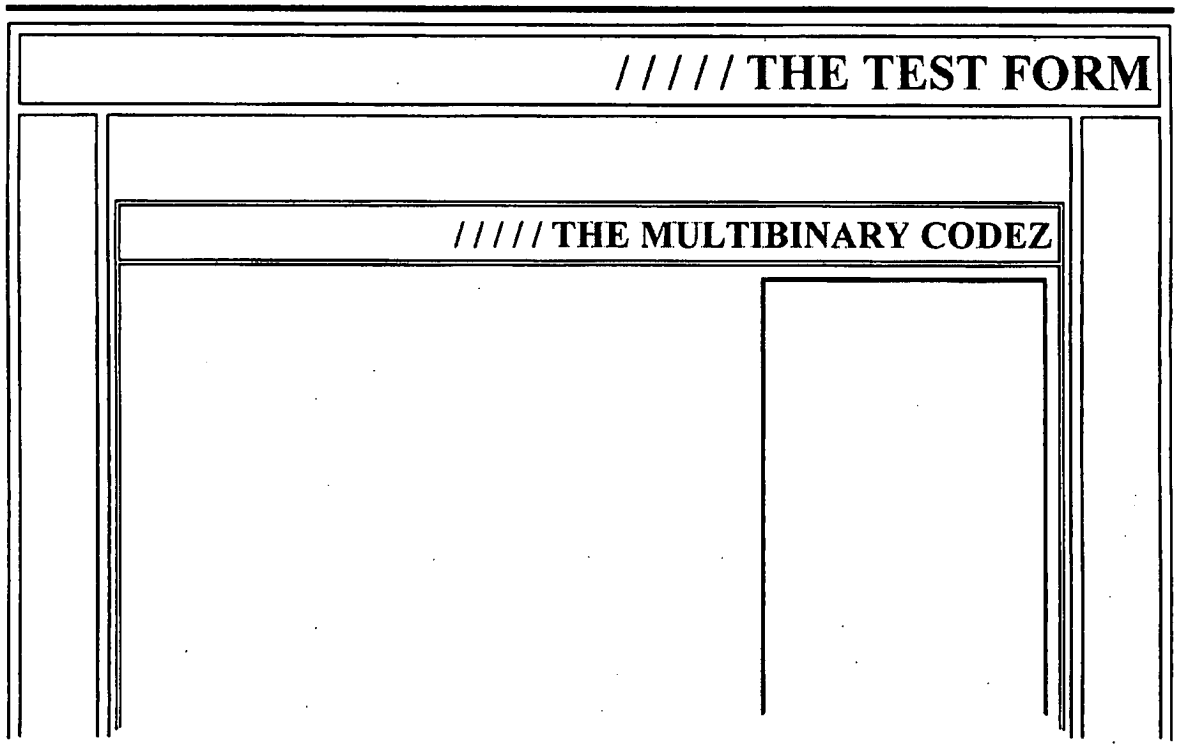
MultiBinary codes and the TestForm for it.
The simple Binary search has only the code but is not featured in the test form since it is a procedure that has not been devised by me

I prefer focusing on what's new, in the Test Forms, instead than on mere implementations of already widely publicized procedures which can be found documented on a variety of books, therefore the Test Form tests only for the multi binary script.

To assess whether the speed is higher, do not ran in *continuous* mode, then consider changing the *subset* amount if the timing seems higher than the traditional loop (the Test Form, in fact, can compare these two timings!).
On the whole you're to discover that about 50% of the times the multiBinary is much faster (on a 200,000 entry long Array and with a subset of 500, it can score 20 milliseconds against 680 of a traditional loop), whereas for the other 50% it can be slower, but not to the same high percentage it can be when it is faster. On the whole, whenever you have to deal with very long Arrays and you're searching for one specific item, you may want to consider the multiBinary like a well pondered bet: when you gain, either you gain a lot (at times much over 300% as you noticed in the example above)

or a little (at least some 10%), and when you lose you lose little (at most 150%. average loss is 30 or 40%). A whole balance seems providing you, when you run it many times on a variety of array lengths and different subsets, with something between a 20% and 35% of overall gain. So worth it, if your task is repetitive and on long Arrays.
Test it:

1. Without continuous mode
2. Changing the Array length
3. Changing the subsets amount
4. Shuffling the positions where the two entry carrying the text "Hallo World" (for which we search for in our simplified Test Form shell) are positioned at.



```
function multiBinary(array, find,
continuous, subset){
if(!array || typeof(array)!="object"
|| !array.length || typeof(find)
=="undefined"){return null;};
if(array.length==1){return (array[0]
==find)?0:null;};
continuous=(continuous)?new Array
(0):null;
subset=(subset && !isNaN(parseFloat
(subset)))? parseFloat(subset):50;
subset=(subset>array.length ||
subset<=0)?array.length:subset;
fragments=Math.round
(array.length/subset);
var indexes=new Array(0);
```

VIEW | Internet Explorer: with comments? »

///// TEST IT

Step 1: produce a LONG array first.
Randomly introduce in it some text to find.

Array length: 30,000

"Hallo World" randomly inserted at pos.:

11201 21007 < Shuffle again

[you can also custom them by hand]

Step 2: set Parameters and run *multiBinary*

continuous? » | **subset »** 50

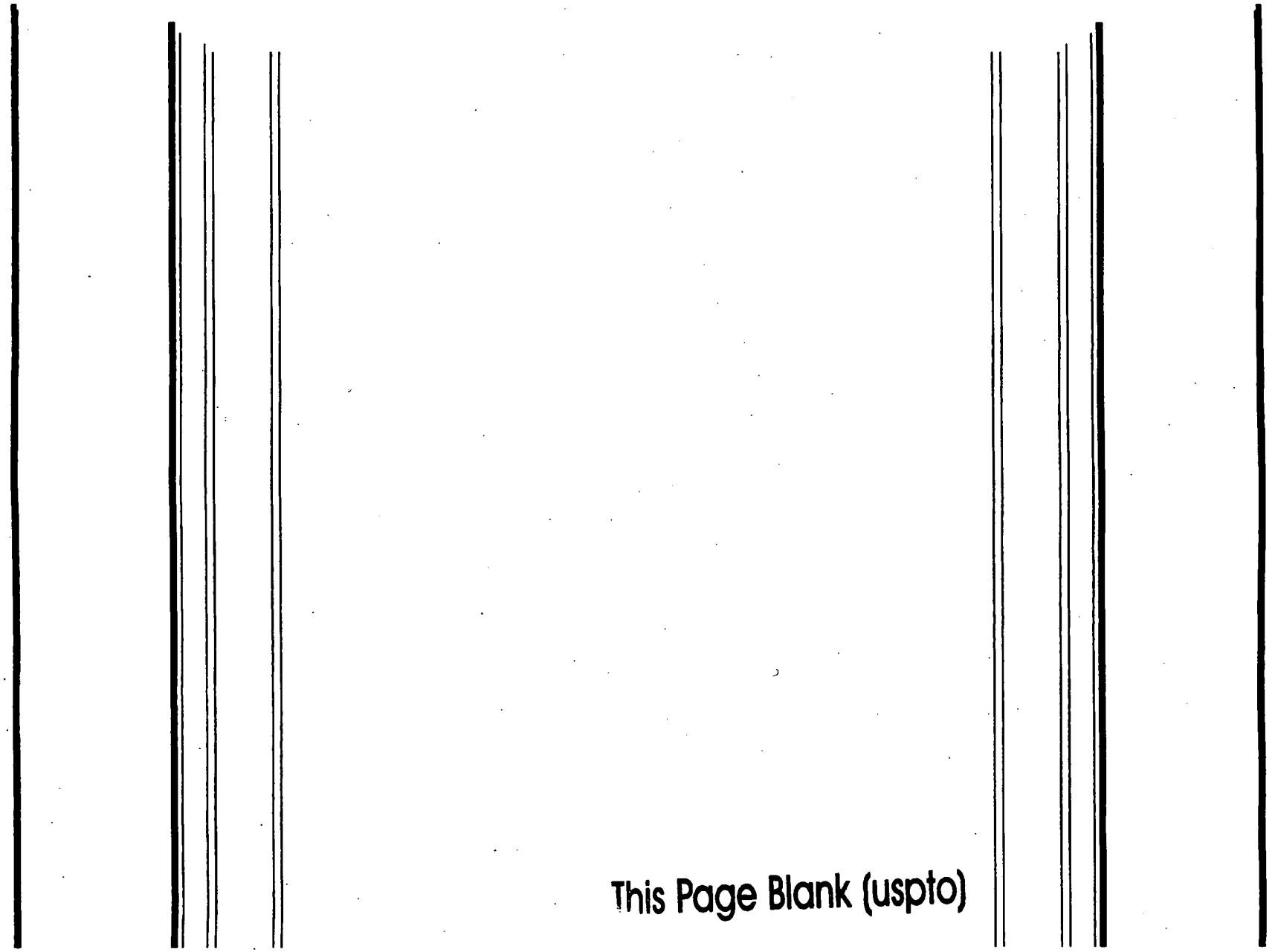
Automatic **Shuffling** on Trigger? »

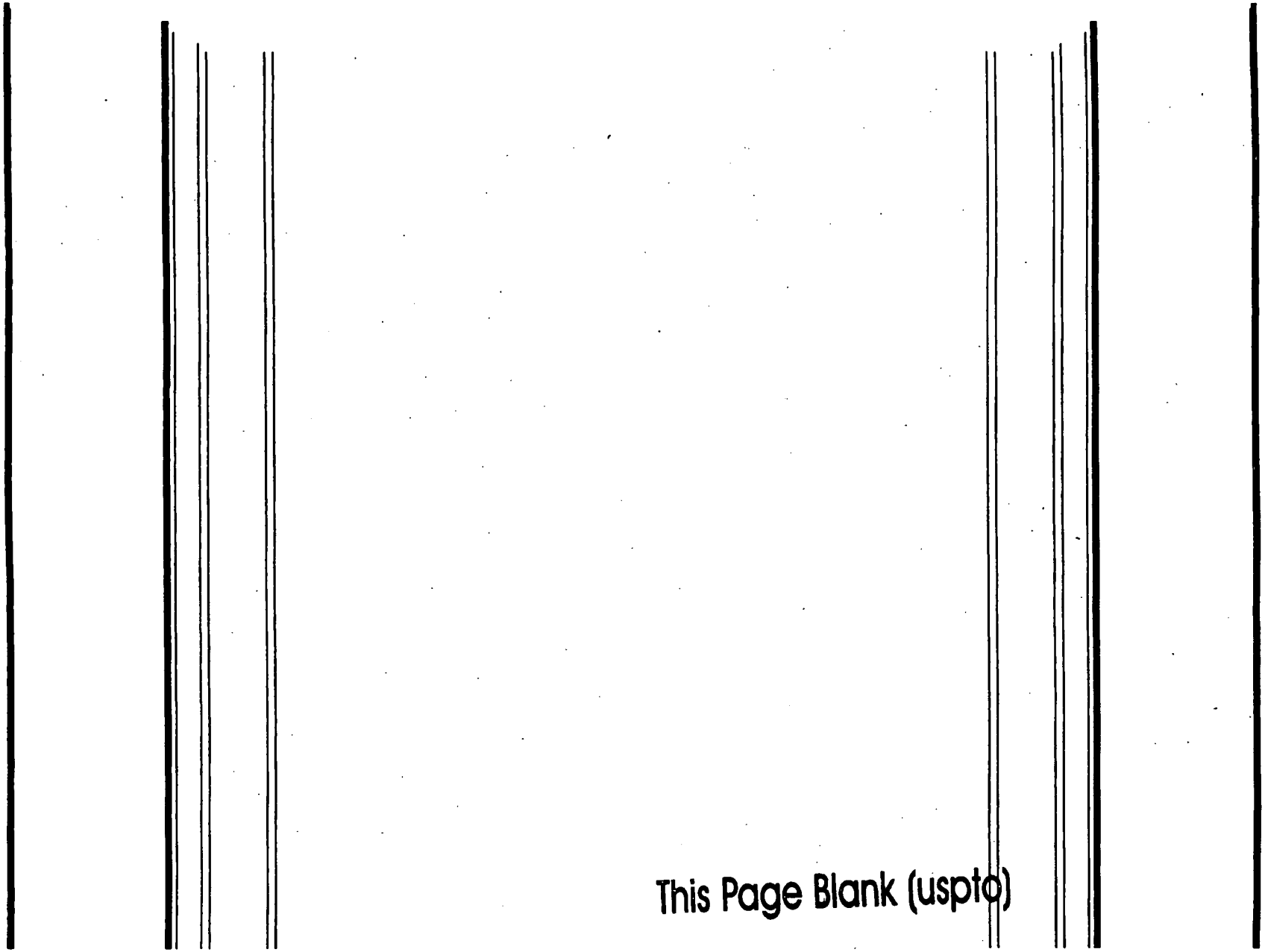
linear loop would take. If you also want these tests being run (would take a few more seconds) as soon as you click the TRIGGER TEST button, please

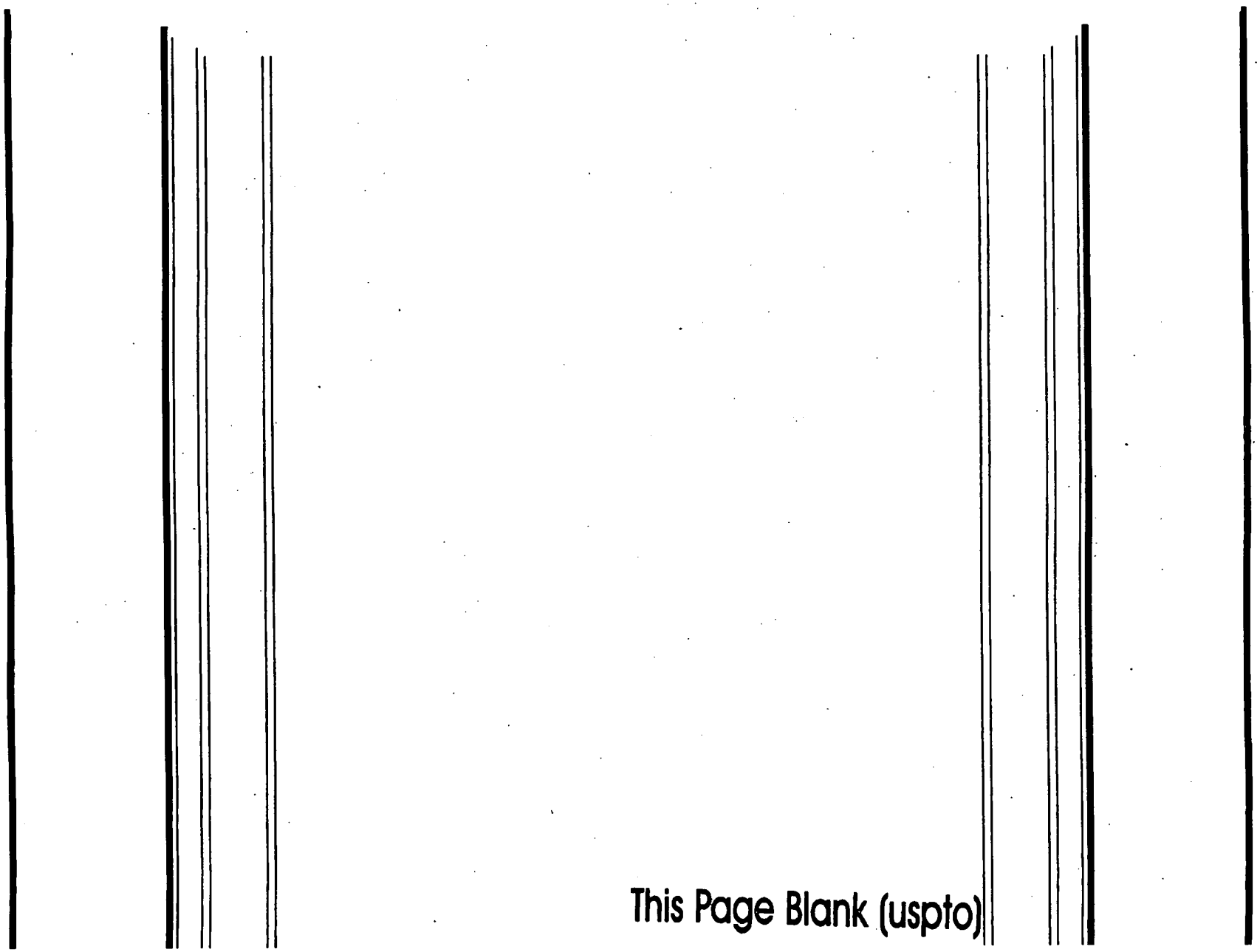
check this »

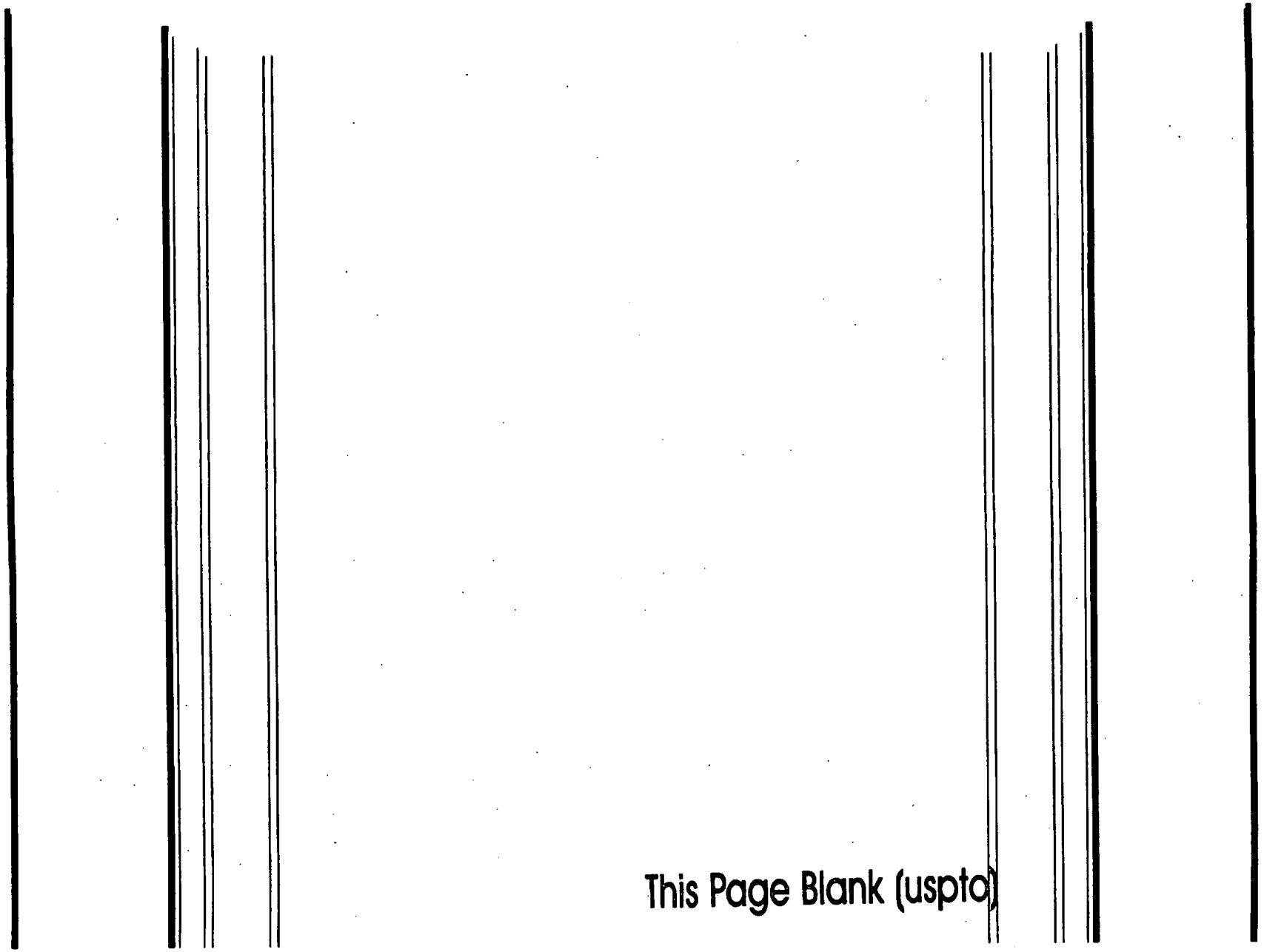
MULTI BINARY score:	SIMPLE LOOP score:
is	is
overall:	overall:

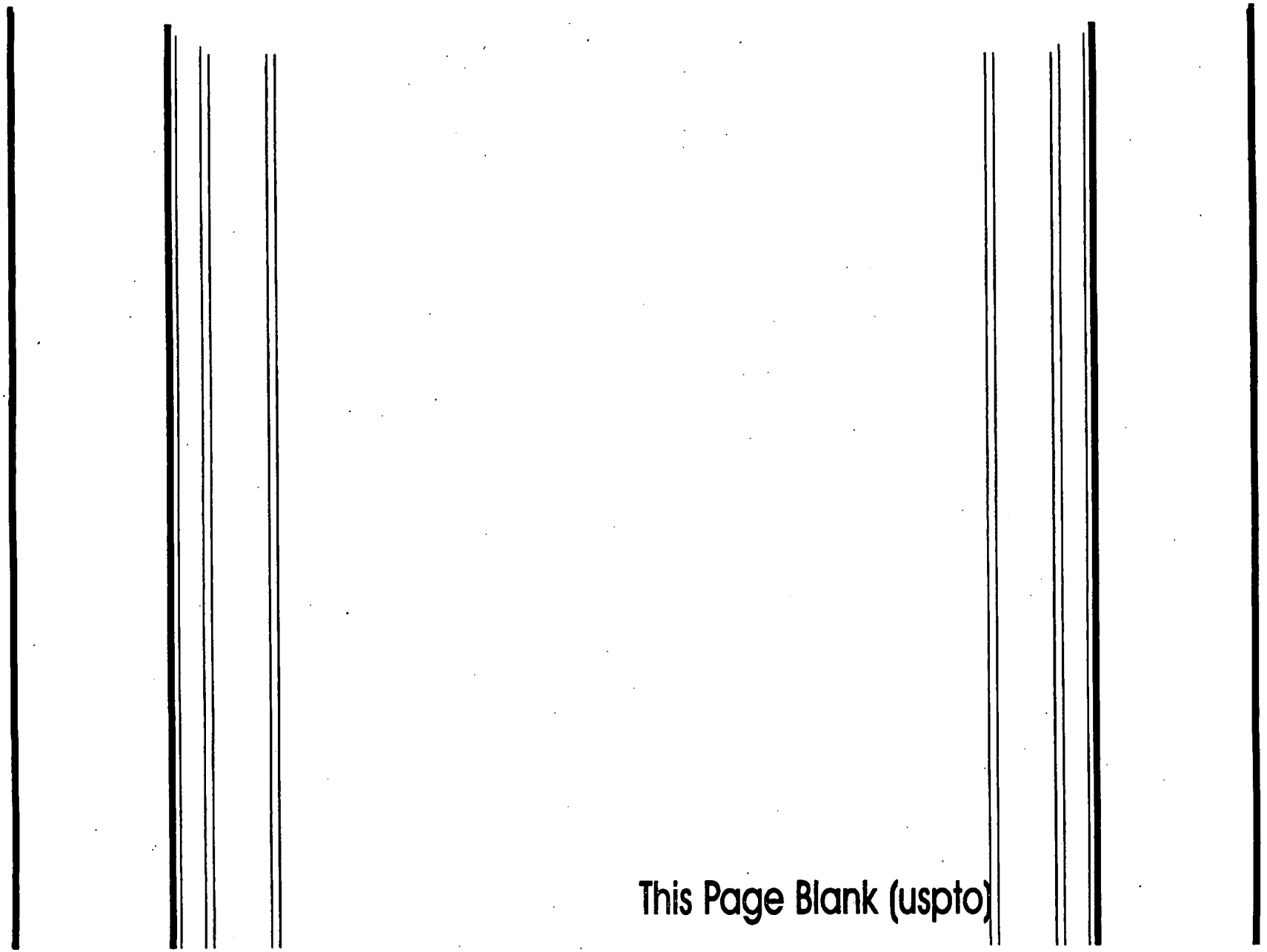
Overall of overall: on Runs:

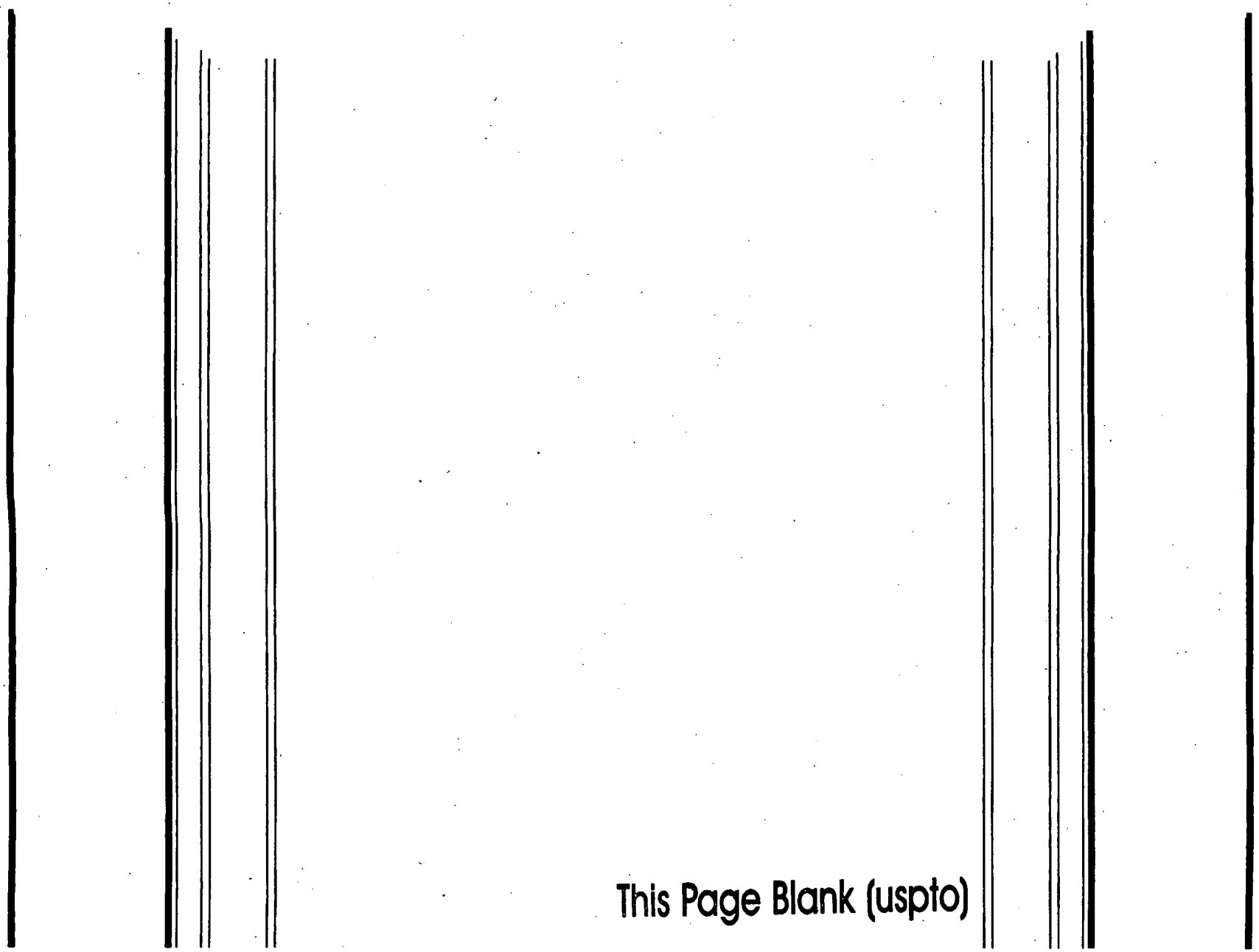


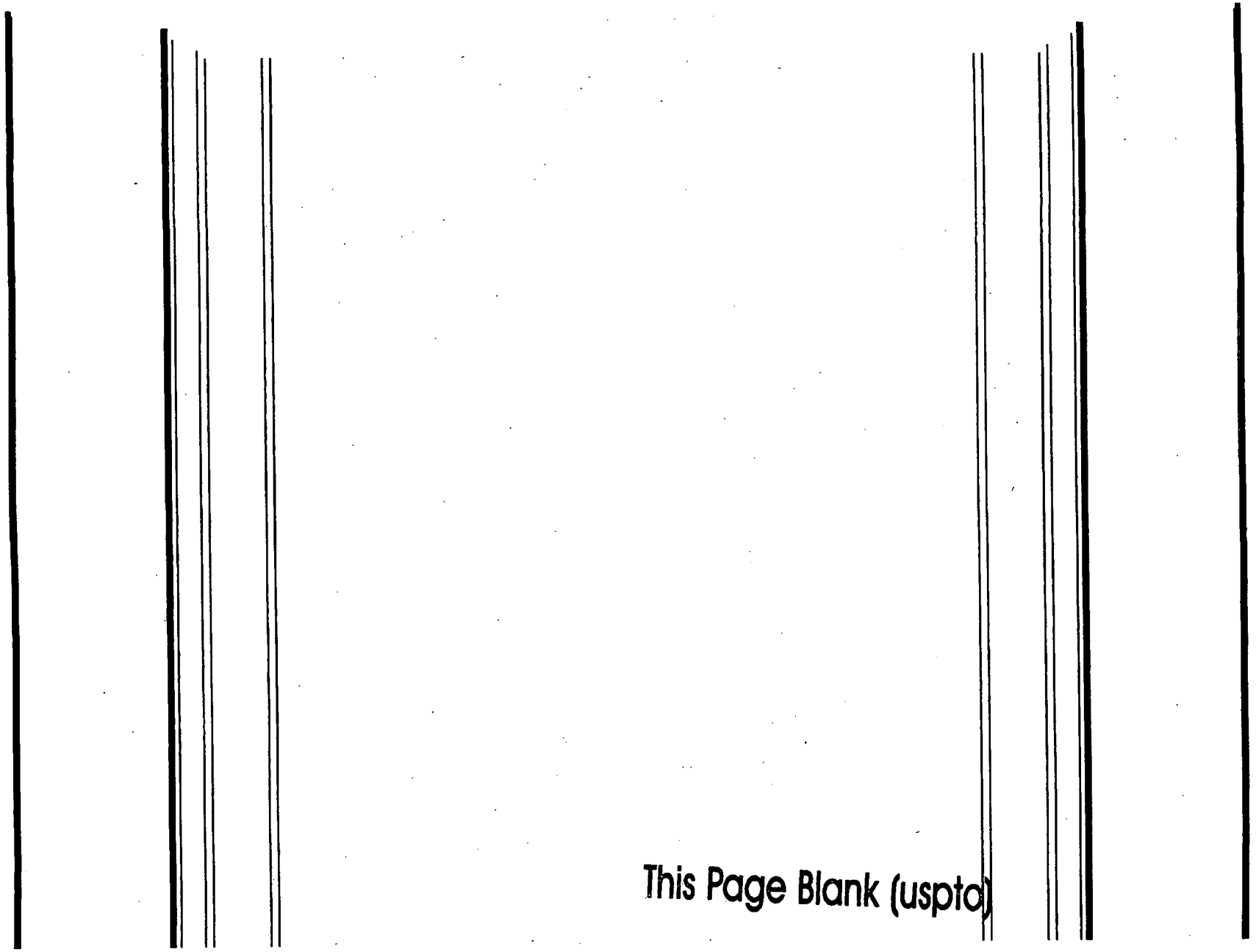


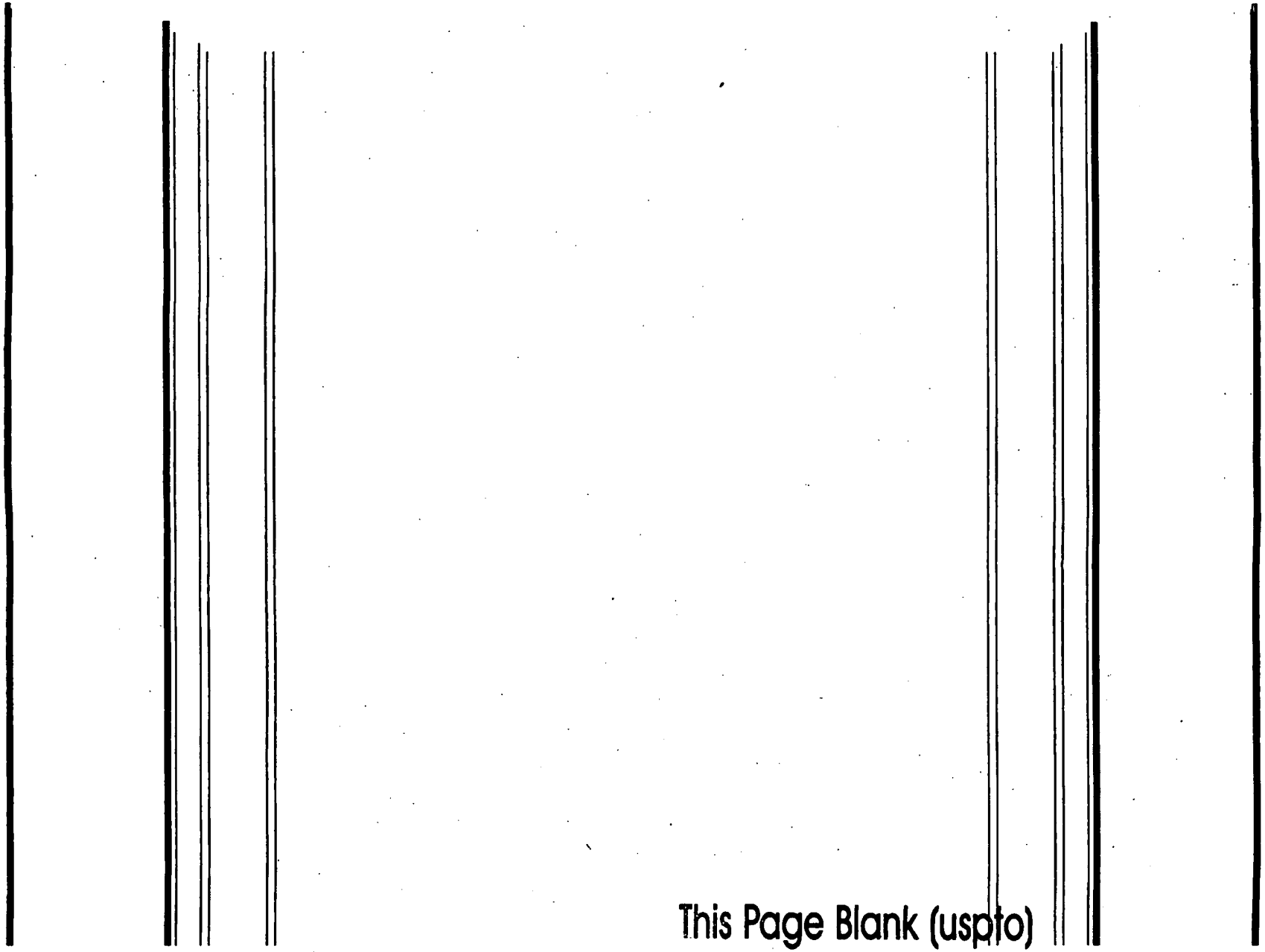







This Page Blank (uspto)





TRADITIONAL BINARY SEARCH

A javaScript implementation of a Binary Search

As I already stated, this code below is not an original one but a mere javaScript implementation of a traditional *Binary Search*: the reason I'm providing you with this is that I believe a website dealing mostly with javaScript could also provide a few mere implementations of scripts originally meant for other languages (moreover, as far as I know no website to date has a binary search *javaScript* implementation. So one more reason); none the less, the meaning of this website is to some degree the reverse of this: providing you with original scripts and algorithms written in javaScript but that as such can be easily translated/implemented in other languages by simple syntax swappings (at least those scripts which are not *Dhtml* geared).
The following implementation has been crafted after the one featured in *Mastering Algorithms in Perl* by *Orwant, Hietaniemi, Macdonald, O'reilly editions*: a complex book that I never succeeded in finishing but that surely features a deep approach to algorithms although *Perl* oriented.

let me stress, for all those using copy and paste snippets and that, having committed no time to face a scripting riddle, and who therefore think each script they find is "such simple a task", that in order to develop "such simple a thing" like a binary search seems, it has taken 20 years. I find a variety of guys, at times relatively affirmed professionals, that on news groups spend their valuable time seriously mocking at beginners (at times even not so much beginners, actually: but the former invariably assume the latter must be, ya see) "oh such simple a thing, and you don't know how to do it!"; I believe they have no clue: as Jon Bentley says, again in *Programming Pearls*:

> «[this codes have been used] precisely in their 1981 *Software tools in Pascal* to move lines within a text editor. Kernighan reports that it ran correctly the first time it was executed, while their previous code for a similar task based on linked lists contained several bugs. This code is used in several text processing systems (...) Ken Thompson wrote the editor and this reversal code in 1971, and claims it was part of the folklore even then.» [chapter 2]

and:

> «while the first binary search was published in 1946, the first binary search that works correctly did not appear until 1962.» [chapter 5]

In order to use this script you must be aware of 6 things:

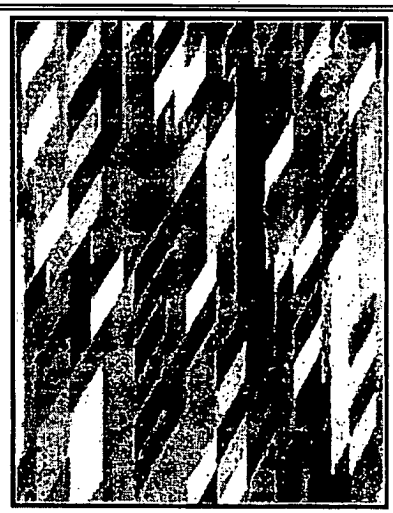
1. A binary search can produce the right results only and exclusively if the inspected Array has been sorted, that is: is in alphabetical or numerical order.
   I still find at times some scripters that although experienced wonder what I mean when I say that simply sorting an Array doesn't wort it right: well, it means that if you just say: Array.sort (), the contents would NOT be arranged AT ALL in the order you may expect: in fact:
   - Capital letters (if they are letter) would all go either before or after the lowercase ones, with the consequence that a capital B may appear listed... before a lowercase a!
   - Numbers starting with the same number would be associated: therefore 1, 2, 14 would not be listed as such but as: 1, 14, 2 !

   To overcame this you must pass to your *sort()* javascript method an argument, which must be a function, to sort your Array in the expected way. All such snippets are available, with further documentation on how to use them, at United Scripters on **this file** (they are the four snippets at top of the page named after the shared string "*highest*").
2. You pass to the function as first argument the **already sorted** array.
3. You pass to the function as second argument the item you want to find a match for. if a string, in between quotes as usual.
4. If an Array contains more than one instance of the given match, the binary searchers have a tendency to return only one *somewhere midway* found instance: if you have an array like [5,5,5,5,5,5], the returned number would be 2: meaning at position 2 (starts counting with zero) number 5 has been found. if you want to inspect the surroundings, you may first find such an instance index, and then make micro loops around such index unless you find a higher or a lower number in order to assess the boundaries of the item. I say this just in case you may be wondering, you see.
5. The function returns either null if no instance found, or a number which is the numerical index of the array where the instance has been pinpointed.
6. Actually I have included an original variation in the binarySearch: since normally you'd have had even a further requirement (namely that the array is not just sorted and sorted right, but

even that the lowest values must be at the beginning and the highest at bottom of it), I slightly modified it so that it can guess on its own whether the highest values are at top or at bottom (say: A-Z or Z-A?) and perform accordingly. otherwise only in the A-Z order a binary search would have worked.
Interesting, don't you think so?

///// THE BINARY SEARCH CODEZ

```
function binarySearch(array, find){
if(!array || typeof(array)!="object"
|| typeof(find)=="undefined" || !
array.length){return null};
var low=0;
var high=array.length-1;
var highOnTop=(array[0]>array
[array.length-1])?1:0;
var middleArray=parseInt
(array.length/2);
//run:
while(low<=high){
var mid=(low+high)/2;
var aTry=(mid<1)?0:parseInt
(mid);/*NS4: parseInt on zero yields
NaN!*/
```



Show

TOP OF PAGE